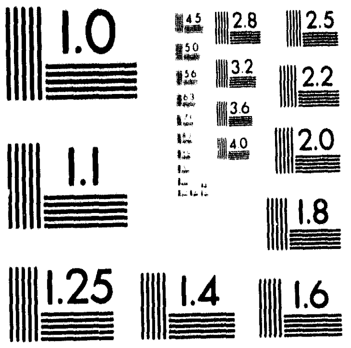
1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

1 of 1

\ ; [illegible] 9106442 /

Proceedings of the Conference on Parallel Architectures and Languages Europe Eindhoven, The Netherlands, June 1991

CHAOTIC LINEAR SYSTEM SOLVERS IN A VARIABLE-GRAIN DATA-DRIVEN MULTIPROCESSOR SYSTEM*

Jean-Luc Gaudiot and Chih-Ming Lin

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-0781

Abstract

Linear systems are important problems in many scientific applications. While asynchronous methods are effective solutions to linear systems, they are difficult to realize due to the chaotic behavior of the algorithms. In this paper, we investigate the implementation as well as the performance of an asynchronous method, namely chaotic relaxation, in our Variable-grain Tagged-Token Data-flow (VTD) System. We compare asynchronous methods with synchronous methods executed on both the fine-grain and the coarse-grain execution models. New high-level data-flow language constructs are introduced in order to express asynchronous operations. A new *firing* rule that deviates from the *single assignment* rule of functional languages is proposed to support the implementation of asynchronous computations in the VTD system. In addition to the conventional *speedup* measure, we then define new performance measurements, called *Growth Factor*, *Scalability Factor*, and *Robustness* to characterize the system performance from the machine and application viewpoints. Simulation results indicate that asynchronous methods are more efficient than synchronous methods and that the coarse-grain execution mode is more efficient that the fine-grain execution mode in our VTD system.

1 Introduction

Linear systems play an important role in many applications such as PDE solvers. Generally, linear systems can be solved by direct or iterative methods. Iterative methods can further be classified as synchronous [9] or asynchronous [3]. While synchronous methods are easy to implement, they do not yield acceptable levels of performance for complex problems, mainly because of the synchronization necessary among the various processes. On the other hand, asynchronous approaches have been found by many researchers [3, 5] to efficiently exploit runtime parallelism. In an asynchronous approach, communication between processes is achieved by reading the dynamically updated variables while each

* This material is based upon work supported in part by the U.S. Department of Energy, Department of Energy Research, under Grant No. DE-FG03-87ER25043.

[illegible]

process continues its execution to update shared variables. Therefore, the chaotic behavior of data in an asynchronous algorithm is very complex. However, while an asynchronous method can be effective in parallel machines and can deliver high performance, it is difficult to implement due to the chaotic behavior of the method itself. From the software perspective, language constructs must be defined to specify the asynchronous method, thereby parallelizing the algorithm. From the hardware point of view, special architecture schemes dedicated to the algorithm need to be developed.

The data-flow principles of execution [2] offer the programmability needed to synchronize at runtime the many parallel processes on a large scale multiprocessor. Instead of relying on the conventional central program counter, the availability of data renders an instruction executable. Asynchronous algorithms have been implemented in data-driven systems, more precisely in *micro-actor-based* data-driven systems [5]. Although the micro approach to asynchronous methods correspond well to the simplicity of data-driven principles, it yields much overhead to respect the functionality of execution.

In this paper, we will first introduce special high-level data-flow language constructs (*Async-Repeat* and *Async-For*) to describe the chaotic behavior in asynchronous algorithms. The scheme to form coarse-grain (macro-actor) data-flow graphs and a specific firing rule in the *Matching Store with Locks* of processors will also be introduced in order to correctly execute the computations of the asynchronous algorithms. In this paper, we are also interested in measuring and comparing the performance of algorithms as well as our VTD system: First, to evaluate the performance of the architecture, the conventional *"Speedup"* measurement will be taken to depict the trend of the performance with larger machine configurations. Second, to estimate the growth of parallelism within an algorithm when the algorithm's complexity has been increased, a new measurement, called *"Growth Factor"*, will be defined to show how suitable an algorithm is for multiprocessor systems. Third, to measure the efficiency of parallel systems in the execution of parallel algorithms, we will introduce another new measurement, called *"Scalability Factor"*, to demonstrate the scalability property of the systems. Finally, we will define *"Robustness"* to indicate the potential performance of the systems.

We shall start our discussion in section 2 by giving a brief introduction to the data-flow principles of execution as well as to asynchronous methods for solving linear systems. In section 3, the Jacobi method and the chaotic relaxation method are described in a High-level data-flow language along with the new languages constructs. The VTD System and the new firing rule for chaotic relaxation and the new performance measurements will be described in section 4. Section 5 will present the results of a deterministic simulation on the system and concluding remarks will be made in section 6.

2 Data-flow Principles and Iterative Solutions for Linear Systems

In this section, we first introduce the data-flow principles of execution and review the essentials of the synchronous and asynchronous linear system solvers which will be evaluated on our VTD system.

2.1 Data-flow Principles

Programmability has been identified as the major issue in the design of large-scale multi-processor systems [1, 2]. Indeed, programmers cannot be expected to be able to schedule and synchronize the hundreds or thousands of tasks that are required to fully utilize the resources of such machines. Therefore, the data-flow model of computation has been introduced to alleviate this problem [1]. Data-flow principles allow runtime synchronization of operations based on their data dependencies. This allows a very large number of different tasks to be scheduled efficiently and transparently.

Data-flow computing is an alternative to the control-flow model. It is inherently parallel, as the execution of an instruction is based upon the availability of its arguments. Data-flow principles can be characterized by two statements: First, operations execute only when all required operands are available. Second, actors are purely functional and execution produces no side-effects. Data-flow programs are represented by directed graphs which consist of actors connected together with arcs. Arcs represent the data dependencies between actors and carry tokens which are the data values passed between actors [2].

2.2 Jacobi Method

The Jacobi iterative method can be derived from a liner system $A \times x = b$ as:

$$x_i^{(k+1)} = \frac{-\sum_{j\neq i, j=1}^{n} a_{ij} x_j^{(k)} + b_i}{a_{ii}} \quad for\ i = 1, \ldots, n\ and\ k \geq 0 \tag{1}$$

where the $x_i^{(0)}$'s are initial estimates of the components of the solution x. By examining the Jacobi iterative method shown above, it can be seen that all the components of the previous (*old*) vector $x^{(k)}$ must be saved before the components of the next (new) vector $x^{(k+1)}$ are computed. Therefore, in this algorithm, an iterative sequence of approximations $x^{(1)}, x^{(2)}, \ldots,\ x^{(n)}$ will be sequentially computed.

2.3 Chaotic Relaxation

In the asynchronous approach, each process continues execution to update the elements in $x_{(i)}$ and communication between processes has been achieved by reading the dynamically updated variables. A subset of asynchronous methods, called *chaotic relaxation schemes*, was introduced by Chazan and Miranker [3] to solve linear systems. In a chaotic relaxation scheme, practical constraints on the asynchronous behavior are imposed. While an asynchronous algorithm imposes no restriction on how "old" a value may be (*i.e.*, how many iterations ago it was produced), chaotic relaxation requires that the updated value of a point be received within a fixed amount of time.

3 From Algorithms to Data-flow Graphs

We have now established two categories of algorithms for linear system solvers that we will implement and evaluate on our VTD system. These algorithms will now be expressed in a high-level data-flow language translated into data-flow graphs.

3.1 The Jacobi Method and Synchronous Constructs

The algorithm is shown in Fig. 1 in SISAL [8]. Line 4 contains the decision to proceed or not with the relaxation at each iteration. The *Relaxation* procedure (lines 5 through 15) performs the relaxation for all of the elements in $X[i]$ and generate new values of vector $X[i]$. The *Convergence Check* procedure (lines 16 through 22) checks all the elements and generates a termination signal back to line 4. Here, we use a stopping criterion evaluated by an L_∞ norm.

In the SISAL program, one should note that the relaxation on each element $X[i]$ under the *for* constructs (lines 5 and 6) can be executed in parallel mainly due to the definition of the language constructs. In the same way, the convergence check on each elements of $X[i]$ and *old* $X[i]$ can be executed in parallel under the *for* constructs in line 16. However, the algorithm itself will be executed in a synchronous manner. In other words, a ***step-by-step*** iteration process will take place.

3.2 Chaotic Relaxation and Asynchronous Constructs

The chaotic relaxation is an approach which is particularly successful in parallel environments. However, new language constructs must be introduced to describe the algorithm.

3.2.1 The Asynchronous Constructs

Asynchronous computations cannot be easily implemented by traditional high-level programming constructs. Therefore, we designed the ***async-repeat*** and the ***async-for*** operators to represent an asynchronous behavior. The main idea of the new ***async-repeat*** and ***async-for*** constructs is to release the synchronization constraints from the *repeat* and *for* constructs in SISAL since then inherently create synchronization points in the body of the loops.

1. **Async-repeat** : This construct allows the procedures inside it to be concurrently evaluated without any synchronization between one another. For example, in the following program, statement (1) and statement (2) can be executed simultaneously and repeatedly as long as the condition $c \leq 100$ remains true:

```
for initial
while c ≤ 100 async-repeat
        a:= old a + 1 ; - - - - (1)
        c:= a + b ; - - - - - - - (2)
return value of c;
end for
```

In the above program, under the asynchronous construct, statement (2) may be executed and its result generated before the completion of the execution of statement (1). In other words, if statements (1) and (2) were executed independently, c may be already larger than 100 (assume $b = 101$ and $a = 0$). This would force termination of the process before a is updated by statement (1). Note that the execution model described above will not be allowed in the conventional *repeat* construct which only executes the two statements one after the other due to the synchronization point imposed by the language construct.

```
define main, jacobi
type OneDim = array[real];
type TwoDim = array[OneDim] ;
function jacobi ( A : TwoDim ; B : OneDim ; N : integer ;
                  returns OneDim )
(1) for initial
        Err := 0 ;
        X := array [1: 0.0, 0.0, 0.0, 0.0] ;
(4)     while Err < N repeat         % convergence check
(5)         X := for i in 0, N        % relaxation on X
                temp1 := for j in 1, N
                    temp2 :=
                    if i ≠ j
                        then A[i,j] * old X[j]
(10)                    else 0.0
                    end if ;
                    returns value of sum temp2
                end for;
                returns array of ( B[i] -temp1 ) / A[i,i]
(15)        end for;                  % generate new X
            Err := for i in 1, N      % generate error norm
                temp3:= if abs(X[i] - old X[i]) < ε
                    then 0
                    else 1
(20)            end if ;
                returns value of sum temp3
            end for ;
            returns value of X
        end for
    end function
```

Figure 1: A SISAL program for Jacobi methods.

2. **Async-for** : While the conventional *for* construct in SISAL allows every index value to be synchronously executed in parallel, the *async-for* construct releases the synchronization between each index value and allows independent execution of index values in parallel. For example, in the following program, the return values of array X does not need to wait until all the new values of each index i become available. Instead, each new value of index i can be updated asynchronously as soon as the value is available and the next computation can be started.

```
for initial
X := async-for i in 0, N
     temp := Y[i] × old X[i] ;
     returns value of temp × temp
end for;
```

While the *async-for* construct allows each index i to be evaluated asynchronously, it should be noted that the operation within the construct corresponds to an infinite loop. It ensures that the computation will proceed until another process (outer loop) terminates the whole execution. The following program is an example which shows that the process under the *async-for* construct will be terminated by the process that is under the *async-repeat* construct.

```
for initial
while condition async-repeat
    X := async-for i in 0, N
        temp := A[i] × old X[i] ;
        returns value of temp × temp
    end for;
    Procedure_two ;
    Procedure_three ;
end for;
```

Overall, under the bodies of the new *async-repeat* and *async-for* constructs, synchronization constraints can be released while the *repeat* and *for* constructs create synchronization points inside the bodies of the constructs. However, in general, the *async-for* constructs will require the *async-repeat* to co-exist in a program. This is because only the *async-repeat* construct can terminate the process under the *async-for* constructs.

3.2.2 The SISAL Programs

Chaotic relaxation can be expressed in SISAL by using the new constructs. First, in order to allow several procedures to be executed asynchronously in parallel, the *repeat* must be replaced by *async-repeat* at the outer loop of these procedures. In Fig. 1, in line 4, the *repeat* should be replaced by *async-repeat.* Therefore, under the *async-repeat* construct, both the *Relaxation* procedure (from line 5 to 15) and the *Termination Check* (from line 16 to 23) can be executed concurrently without any dependency between each other. Second, in order to allow each index value, which is under the *for* construct, to be executed asynchronously in parallel, the *for* must be replaced by *async-repeat* at the beginning of the procedure. In line 5, we replace *for* by *async-for.* Therefore, inside the *async-for* construct, each index value can concurrently proceed the execution of the computation without waiting for other values which are executing the same function.

4 VTD System and Performance Measurements

While the macro-actor concept is a solution which reduces overhead in fine-gain computations, the architecture must be able to execute actors of varying sizes. Our Variable-grain Tagged-token Data-flow (VTD) system has therefore been designed for this purpose. A new firing rule in the VTD system is also proposed to guarantee the proper behavior of chaotic relaxation and to achieve efficient computations. To characterize the performance in the VTD system, new performance measurements are defined along with the conventional performance measurements.

4.1 The VTD System

The VTD system consists of a set of identical Processing Elements (PEs) connected by a hypercube (message-passing) communication network. A single PE consists of 4 units : Matching Store Unit, Instruction Fetch Unit, ALU, Token Formatting Unit [5].

4.2 The Matching Store with Locks

In chaotic relaxation, due to the asynchronous iterations at each grid point, the value of each grid point must be saved for the relaxation of other grid points. In order to guarantee the proper behavior of the chaotic actors, we introduce the notion of **locks** at the inputs of the actors. In other words, we create *locks* inside the matching store for the firing of an actor. Note that the implementation of *locks* in actors corresponds to the *Async-repeat* and *Async-for* constructs of the high-level language. The *locks* will be attached to the input actors of a subgraph. These actors represent the processes that can be executed asynchronously under the *Async-repeat* and *Async-for* constructs.

Under the new firing rule, when an actor is fired, the input tokens remain in the input lock until the next input token is received. In this fashion, the incoming token will replace the stored value and will activate once more the actor. Fig. 2 shows the step-by-step the operation of the new firing rule of an actor along with the *matching store with locks*:
1. Initially, when either token A or token B (A and B have the same tags) comes into the actor F, it will be *locked* inside the actor.
2. When the partner token arrives, actor F will be fired and will produce an output token.
3. After firing actor F, both input tokens remain *locked* inside the actor.
4. When another token C is later received by the actor, the actor is fired with the *locked* token on the other port and the new value on the first port. The incoming token will remain locked in the actor. Note that it overwrites the previous token value.

4.3 Performance Measurements

Many measurements of system performance, such as speedup and system utilization, have been used to evaluate multiprocessor systems in the past. However, these measurements do not clearly indicate the effectiveness of architectures as well as application programs

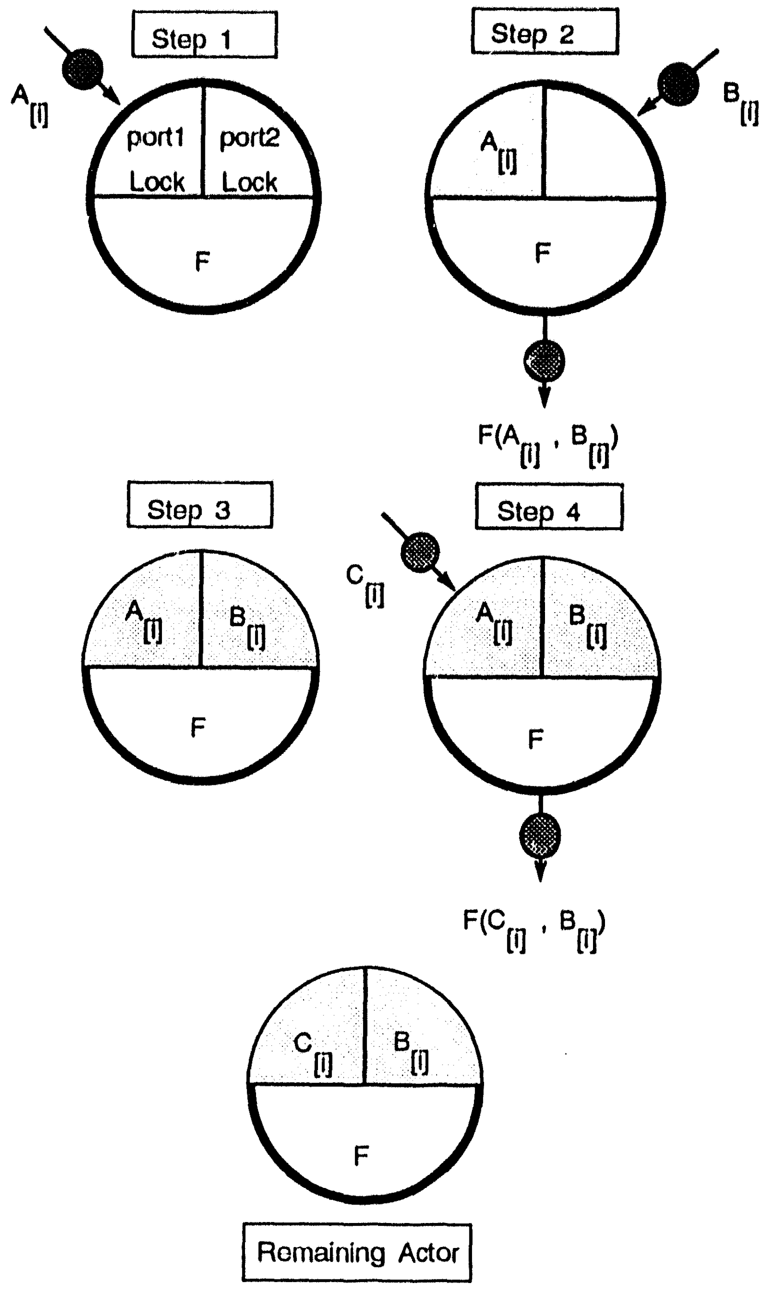

Figure 2: A New Firing Rule with locks in Matching Store.

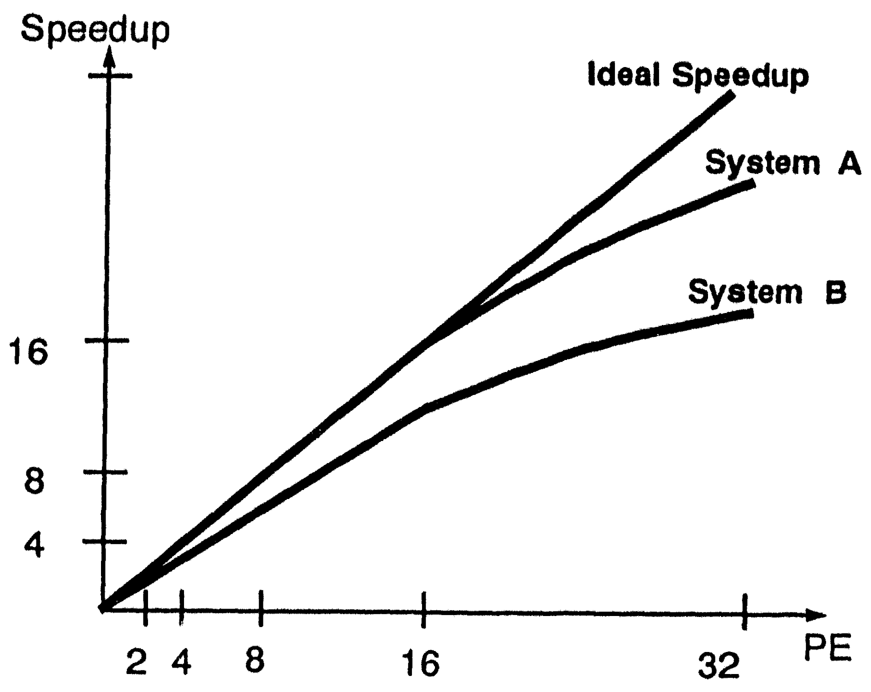

Figure 3: Speedups with Various PEs.

because there is no indication of how much of the performance is due to the architectures and how much of it is due to the applications. Indeed, the speedup should be measured by scaling the problem to the number of processors, not by fixing problem size. An explanation of misuses of Amdahl's speedup formula has been demonstrated in [7]. Therefore, when multiprocessor systems are evaluated, both parallel algorithms and parallel architectures are required to achieve high performance. For instance, a parallel machine cannot deliver high efficiency in executing a sequential program due to the lack of parallelism within the program. On the other hand, a parallel algorithm cannot guarantee high performance in a multiprocessor system if the system cannot exploit the parallelism involved in the program. Clearly, what we need is a better performance measurement to reflect the degree of exploited parallelism resulting from algorithms as well as the ability of the architectures to utilize such parallelism.

In this paper, we are interested in measuring and comparing the performance of algorithms as well as our VTD system: First, to evaluate the performance of the architecture, the conventional ***Speedup*** measurement is taken to depict the trend of the performance with larger machine configurations. Second, to estimate the amount of the growing parallelism within an algorithm when the algorithm's complexity has been increased, a new measurement, called *Growth Factor*, is defined to show how suitable of an algorithm is for multiprocessor systems. Third, to measure how efficient of parallel systems in executing parallel algorithms, we introduce a new measurement, called *Scalability Factor*, to demonstrate the scalability property of the systems. Finally, we define the ***Robustness*** to indicate the potential performance of the systems.

1. **Speedup :** Speedup has been conventionally defined as the ratio of the execution time of an Application (AP) on N Processing Elements (PEs) to the execution time of the same application on a single PE :

$$Speedup\ (AP,\ PE(N)) = \frac{Exe.\ time\ of\ AP\ on\ one\ PE}{Exe.\ time\ of\ AP\ on\ N\ PEs}$$

Under this definition, the *ideal* speedup of an architecture is N when there are N PEs in the system. In other words, if a speedup cure is closer to the line of ideal speedup, then the architecture is considered a better parallel system. For instance, Fig. 3 shows that system "A" performs better than system "B" in term of system "A" having a speedup curve closer to the line of ideal speedup. However, by this definition, it is only shown how the execution time can be reduced in various system configurations while the amount of complexity in the application remains unchanged. However, this does not show the suitability of an algorithm for multiprocessor systems. In other words, the speedup curves only demonstrate the machine domain performance without considering the application aspect.

2. **Growth Factor :** Before the speedup in a system is measured, how well an application can perform in parallel systems must be studied. The growth factor shows how much parallelism changes when the complexity of an algorithm is changed. Here, the complexity of an algorithm refers to the number of operations needed to execute the algorithm. For example, the inner product of vectors $V(a_1, a_2, a_3, ..., a_m)$ and $U(b_1, b_2, b_3, ..., b_m)$ has a complexity of $O(m)$. The growth factor therefore is defined as the ratio of the execution time of an Application (AP) with a complexity $(M \times m)$ on a fixed number of n PEs to the execution time of the same application with a complexity of m on the n PEs.

$$Growth\ Factor\ (AP(Mm),\ PE(n)) = \frac{Exe.\ time\ of\ Mm\ AP\ on\ n\ PEs}{Exe.\ time\ of\ m\ AP\ on\ n\ PEs}$$

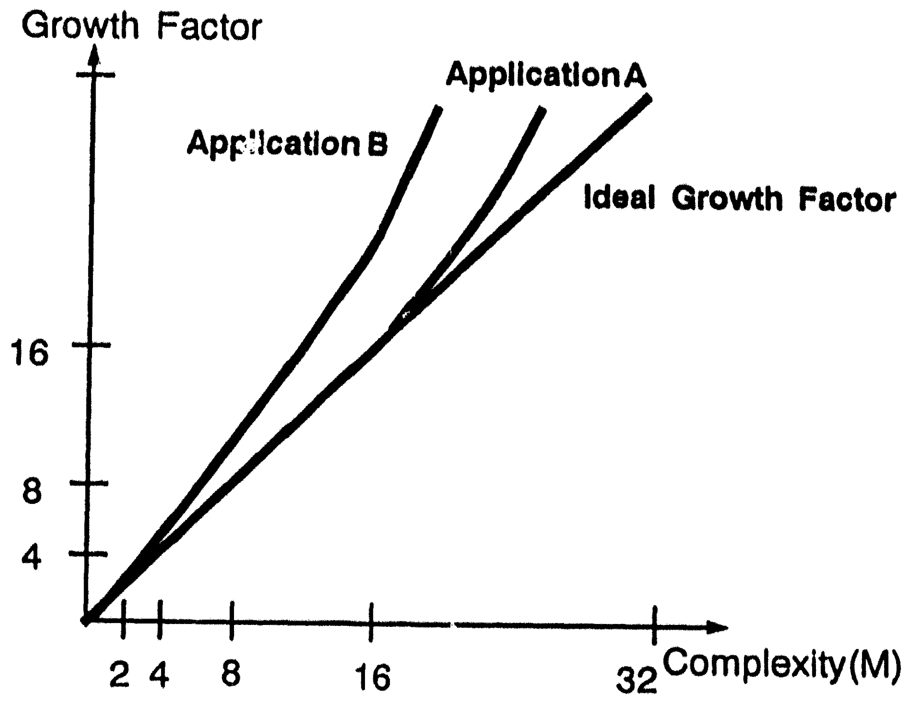

Figure 4: Growth Factors with Various Complexity (M).

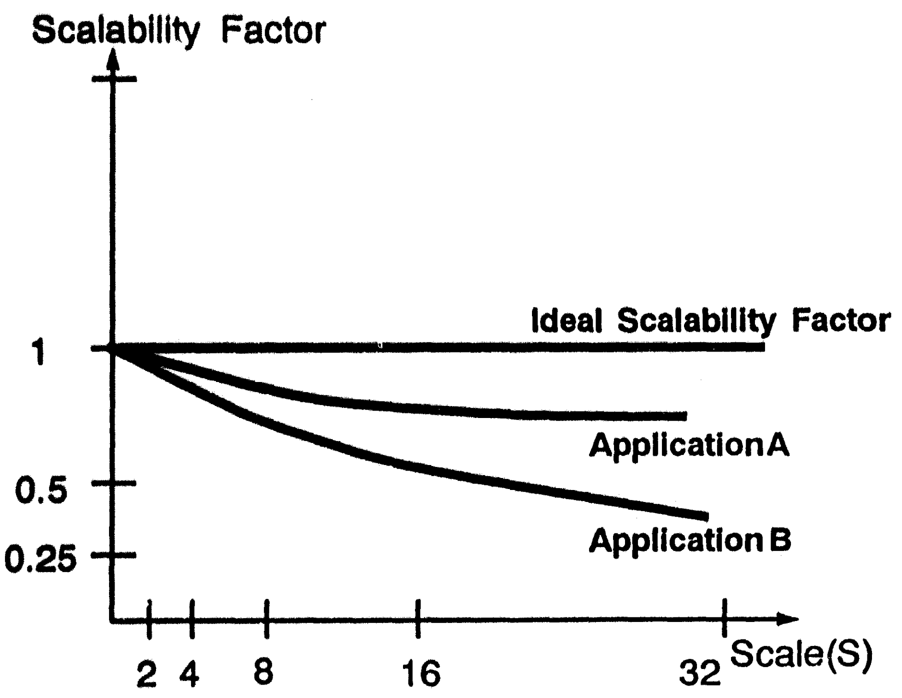

Figure 5: Scalability Factors with Various Scales (S).

Ideally, a perfectly parallel algorithm should have a growth factor proportional to the increasing rate of its complexity (M). For example, a vector to vector multiplication is a perfectly parallel statement that the amount of parallelism increases at the same rate as the vector length (complexity). Therefore, if an application has a curve of growth factor close to the line of ideal growth factor, it is considered a better parallel application. For example, in Fig. 4, application "A" is a better parallel application than application "B" because the curve of growth factor in "A" is closer to the line of ideal growth factor.

3. **Scalability Factor :** The performance of multiprocessor systems should also be measured by comparing the execution time of large problems with that of small problems on single processor systems. In other words, the complexity in the applications should be increased while the size of the machine configuration is increased. The scalability factor is defined as the ratio of the execution time of an Application (AP) with complexity (m) on n PEs to the execution time of the same application with complexity $S \times m$ on $S \times n$ PEs.

$$Scalability\ Factor\ (AP(Sm),\ PE(Sn)) = \frac{Exe.\ time\ of\ m\ AP\ on\ n\ PEs}{Exe.\ time\ of\ Sm\ AP\ on\ Sn\ PEs}$$

If an algorithm has an ideal growth factor and a system has an ideal speedup, then the scalability factor should remain a constant for various values of N. In other words, a perfectly parallel algorithm with a large complexity on a large perfectly parallel system configuration should require the same execution time as it would with a small complexity

Problem Size = 16 × 16				
System Size	Chaotic(Macro)		Chaotic(Micro)	
number of PEs	exe. time	speedup	exe. time	speedup
1 PE	108291	1	108990	1
2 PEs	56690	1.91	54430	2.002
4 PEs	26999	4.01	27174	4.01
8 PEs	13548	7.99	14840	7.34
16 PEs	8050	13.45	10538	10.34
32 PEs	6867	15.76	9708	11.22

TABLE 1 : Execution Time and Speedup in Chaotic Relaxation.

on a small system configuration. However, due to the fact that most algorithms and systems are not perfectly parallelized, the actual scalability factors will fall below the line of ideal scalability factor. Fig. 5 shows that the closer the curve is to the ideal line, the easier it will be to scale up the application/system configuration combination.

4. **Robustness:** The robustness property of a system can actually indicate its potential performance [6]. The robustness is defined as the ratio of the execution time of an Application (AP) with a complexity ($R \times m$) on one PE to the execution time of the same application with a complexity of $R \times m$ on the $R \times n$ PEs.

$$Robustness\ (AP(Rm),\ PE(Rn)) = \frac{Exe.\ time\ of\ Rm\ AP\ on\ one\ PE}{Exe.\ time\ of\ Rm\ AP\ on\ Rn\ PEs}$$

Essentially, robustness is an indication of how well the architecture/execution model will scale up when machine sizes and problem sizes are increased. In fact, one of the most important parameters in evaluating a multiprocessor system is to observe the system performance with various problem sizes. We thus express the performance of an architecture by showing the robustness in a large number of PEs.

5 Simulation Results

Once the Jacobi method and chaotic relaxation have been programmed and compiled into data-flow graphs. The execution of the graphs in the VTD system can be verified by a deterministic simulation in both micro-actor (fine-grain) and macro-actor (coarse-grain) execution models.

5.1 Simulation Results

The execution of the Jacobi method and chaotic relaxation to solve various sizes of linear systems with the termination criterion $||x^{(k)} - x^{(k-1)}||_\infty < 10^{-3}$ have been simulated. From the simulation results, several statistics and observations have been obtained:

1. **Speedup** : The speedup measure has been defined in the previous section. The reports of the speedups in various system sizes for both chaotic relaxation and the Jacobi method are attached in Tables 1 and 2, while Fig. 6 shows the trend of the speedups with

Problem Size = 16 × 16				
System Size	Jacobi(Macro)		Jacobi(Micro)	
number of PEs	exe. time	speedup	exe. time	speedup
1 PE	79924	1	92203	1
2 PEs	42112	1.89	49399	1.86
4 PEs	23109	3.45	27901	3.30
8 PEs	13640	5.86	18219	5.06
16 PEs	9759	8.18	14470	6.37
32 PEs	9244	8.64	13971	6.59

TABLE 2 : Execution Time and Speedup in the Jacobi Method.

increasing PEs for the two different relaxation methods.
Observation: The results indicate that the speedup in chaotic relaxation is better than the speedup of the Jacobi method in both macro and micro execution modes. In chaotic relaxation, a superlinear speedup can be sometimes observed due to the nondeterministic property of the algorithm itself. Indeed, the random sequence of relaxations may lead to a faster convergence in multiprocessor systems. This feature is confirmed in Table 1: the speedups in a 4 PE system for both macro and micro execution of chaotic relaxation can be as high as 4.01

2. **Scalability Factor**: The scalability factor of a system was defined in the previous section. We exploit the trend of scalability factors in different problem sizes with various system configurations. We start with the matrix size equal to 8×8 and the machine size equal 8 PE, then 16×16 in 16 PEs, 32×32 in 32 PEs, and 64×64 in 64 PEs. The report is shown in Table 3 and the curves are shown in Fig. 7.
Observation: The results show that the chaotic relaxation in the macro execution mode of the VTD system has the best scalability factor while the Jacobi methods in the micro execution mode has the worst scalability factor. However, one should note that the increasing rate of the machine size from 8 PEs to 16 PEs does not equal the increasing rate of the complexity of the algorithms with a matrix size from 8×8 to 16×16. Therefore, we only compare the relative performance of different algorithms in various execution modes, instead of comparing it with the ideal scalability factor.

3. **Robustness**: The robustness of a system was defined in the previous section. We exploit the trend of "speedups" in many different problem sizes with various system configurations. We start with the matrix problem size from 8×8 up to 64×64 and the machine size from 1 PE to 64 PEs. The report is shown in Table 4 and the curves are shown in Fig. 8.
Observation: In the results, we know that there are almost linear increasing speedup curves for the two methods in each operation mode. This is a very promising feature for data-driven multiprocessor systems. Indeed, the robustness property of data-flow architectures can guarantee the performance in multiprocessor systems for various problem sizes. For example, from Table 4, the speedup of chaotic relaxation for 64×64 problem size can reach up to 52 in a 64 PEs system with the macro execution mode.

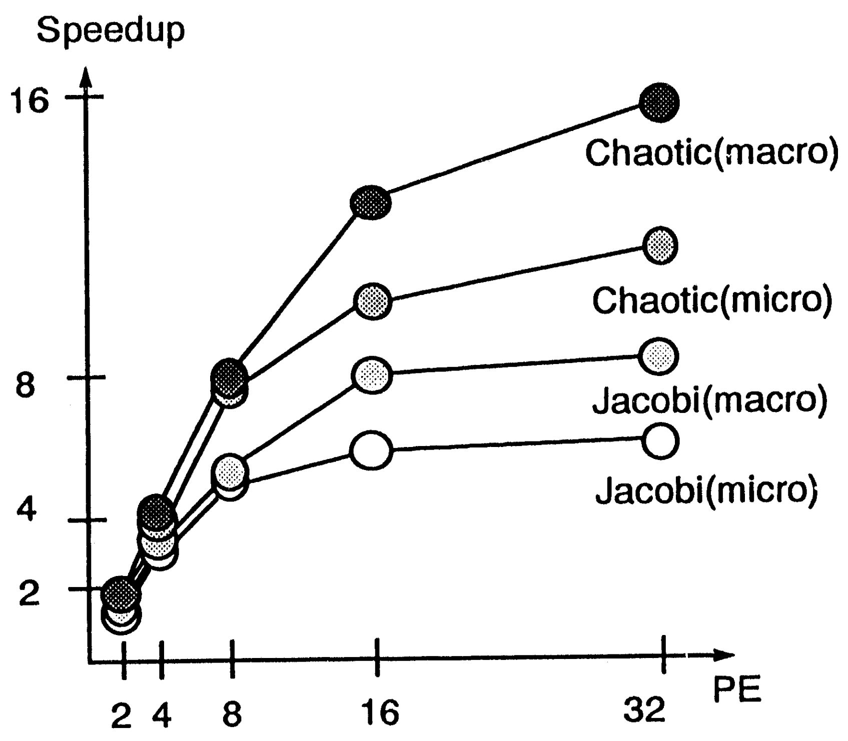

Figure 6: Speedup with Problem Size : 16 × 16.

Scalability Factors					
Number of PEs	Problem Size	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
8 PE	8 × 8	1	1	1	1
16 PEs	16 × 16	0.416	0.409	0.383	0.372
32 PEs	32 × 32	0.278	0.262	0.238	0.226
64 PEs	64 × 64	0.153	0.132	0.121	0.114

TABLE 3 : Scalability Factors in the VTD System with Differents Algorithms.

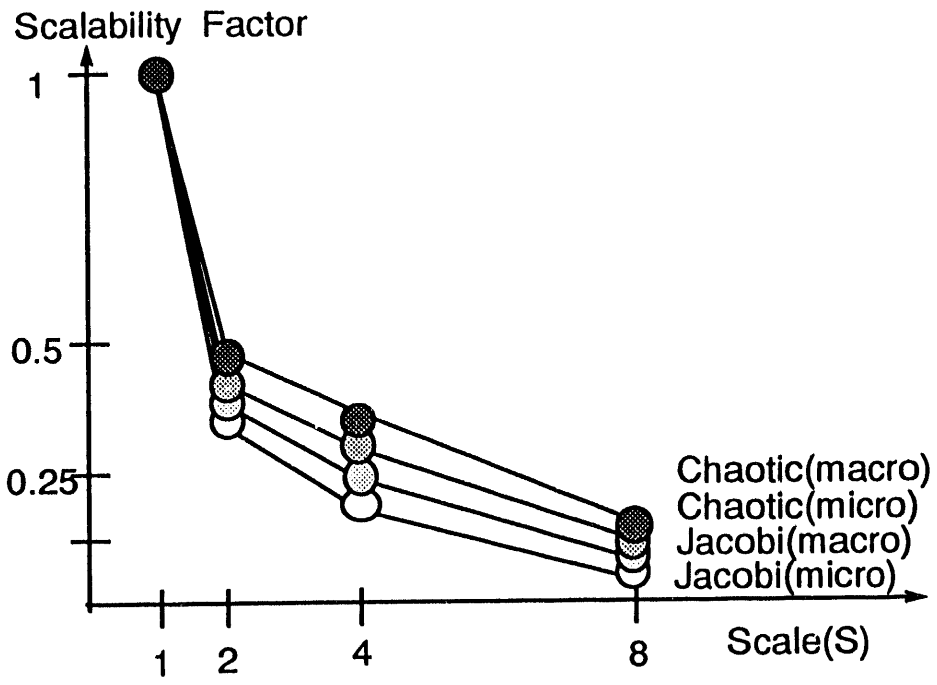

Figure 7: Scalability Factors in the VTD System.

Speedups of Various Problem Sizes					
Number of PEs	Problem Size	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
8 PE	8 × 8	7.15	5.54	4.28	3.45
16 PEs	16 × 16	13.45	10.34	8.18	6.37
32 PEs	32 × 32	26.26	20.30	16.05	12.21
64 PEs	64 × 64	52.15	39.77	31.49	23.70

TABLE 4 : Robustness in Data-flow Architectures.

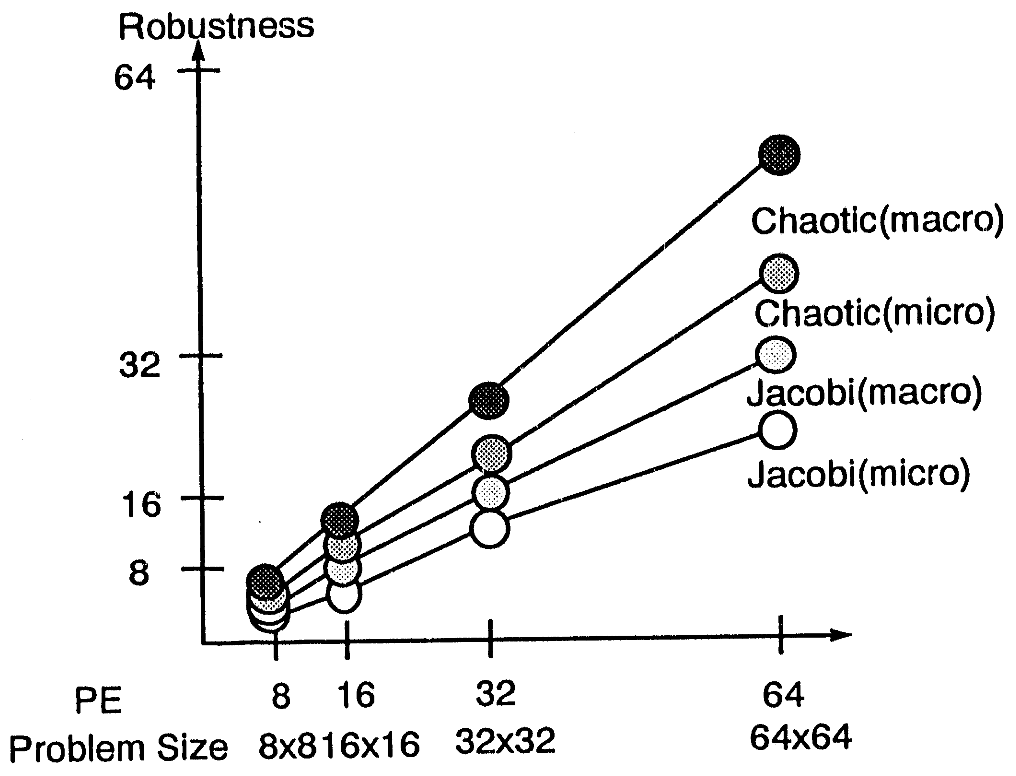

Figure 8: Robustness curves in Data-flow Architectures.

6 Conclusions

In this paper, we have demonstrated how synchronous and asynchronous linear systems solvers could be described in a high level data-flow language (SISAL) and implemented on the Variable-grain Tagged-token Data-flow (VTD) multiprocessor system in both micro and macro execution models. The conventional Jacobi method and chaotic relaxation were chosen for their known inherent parallelism of execution. While the "*conventional*" principles of the U-interpreter were used in the graph construction of the Jacobi method, chaotic behavior could not be easily realized in this model of interpretation. We therefore proposed a new scheme for the implementation of chaotic relaxation: the "*Matching Store with Locks*" scheme proceeds with the execution to detect any change on the input arcs, instead of allowing execution upon arrival of a matched token set. The new defined performance measurements *Growth Factor*, *Scalability Factor*, and *Robustness* have also characterized the system performance more precisely, besides the traditional *speedup* performance measurement in multiprocessor systems.

References

[1] Advanced Topics in Data-flow Computing. Edited by J.L. Gaudiot and L. Bic, Prentice Hall, 1990.

[2] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessors: the data-flow solution. *Technical Report* LCS/TM-241, Laboratory for Computer Science, MIT,September 1983.

[3] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Application*, pages:199–222, 1969.

[4] J-L. Gaudiot and M.D. Ercegovac. Performance evaluation of a simulated data-flow computer with low resolution actors. In *Journal of Parallel and Distributed Computing*, November 1985.

[5] J-L. Gaudiot, C.M. Lin, and M. Hosseiniyar. Solving partial differential equations in a data-driven multiprocessor environment. In *Proceedings of the 15th International Symposium on Computer Architecture*, Honolulu,Hawaii, May 1988.

[6] J-L. Gaudiot and Y.H. Wei. Token relabeling in a tagged token data-flow architecture. *IEEE Transactions on Computers*, September, 1989.

[7] J. Gustafson. Reevaluating Amdahl's law. *Communication of the ACM*, May 1988.

[8] J.R. McGraw and S.K. Skedzielewski. SISAL: *Streams and iterations in a single assignment language*, language reference manual, version 1.2. *Technical Report* M-146, Lawrence Livermore National Laboratory, March 1985.

[9] R. S. Varga. *Matrix iterative analysis*. Prentice Hall, 1962.